Arhitectura Sistemelor de Calcul
Laborator - Partea 0x01

Bogdan Macovei
Ruxandra Bălucea
Cristian Rusu

Noiembrie 2022

Cuprins

1 Stiva

Stiva este o regiune dinamică în cadrul unui proces care funtioneaza pe principul LIFO (Last In – First Out). La fiecare apel de funcție, este alocată o zonă de memorie pe stivă în care vor fi stocate **argumentele funcției, variabilele locale** și **adresa de retur**. La fiecare alocare stiva va crește "în jos", spre adrese mici, așa cum este prezentat și în figura 1.

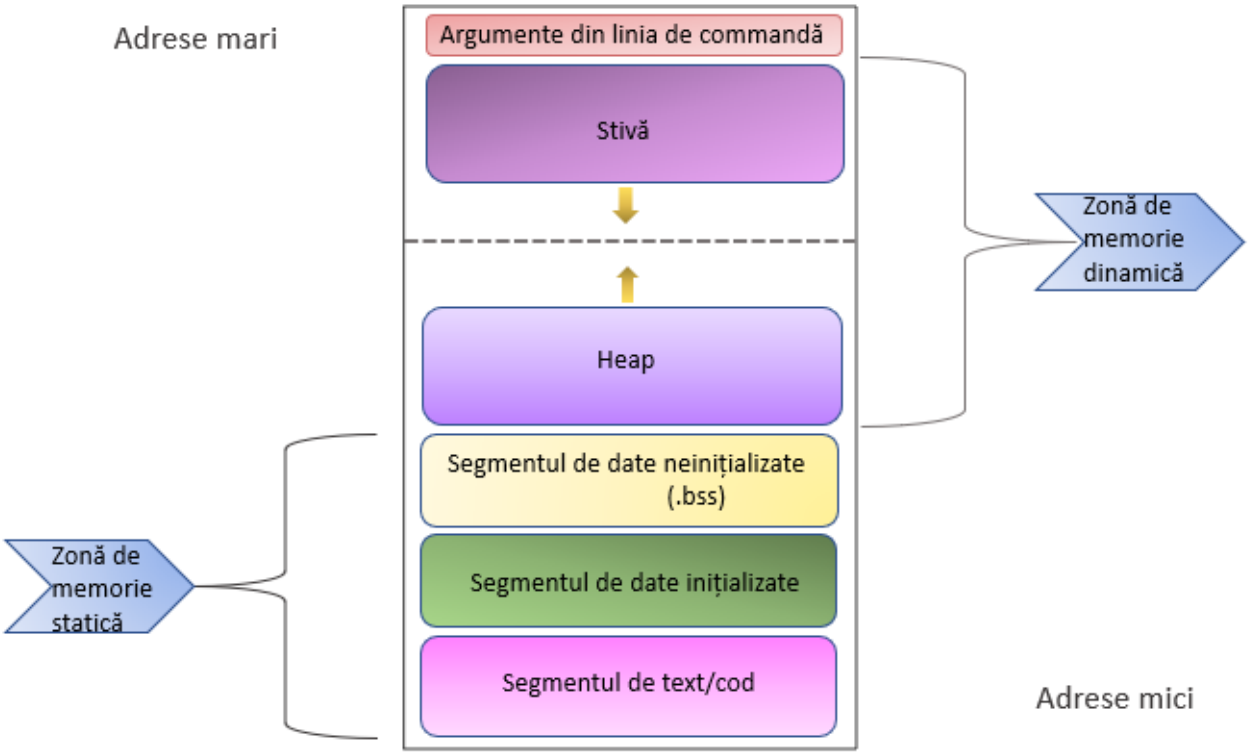

Figura 1: Spațiul de adresă al unui proces

Pentru operarea cu stiva, exista doua instructuni in x86:

```
push operand
```

unde **operand** poate fi o constanta, un registru sau o adresa de memorie. Este folosita pentru adaugarea pe stiva;

```
pop operand
```

unde **operand** poate fi un registru sau o adresa de memorie. Este folosita pentru scoaterea de valori de pe stiva.

Observatie: Pot fi adaugate pe stiva doar **word**-uri și **long**-uri.

Stack pointer-ul %esp este registrul care indică întotdeauna spre vârful stivei și prin care se efectueaza, de fapt, push-urile si pop-urile. De exemplu, la fiecare **pushl** efectuat, %esp scade cu 4 (stiva crește spre adrese mici) si este salvata o valoare relativ la adresa de memorie indicata de %esp. Pentru fiecare **popl** efectuat, valoarea din varful stivei este depozitata in operandul instructiunii, iar %esp creste cu 4.

Urmarind scrierea studiata in laboratorul trecut a(b, c, d) , valoarea efectiva din varful stivei este data de **0(%esp)**. In plus, trebuie remarcat ca varful stivei va indica intotdeauna spre ultimul byte al ultimului element adăugat pe stiva.

Base pointer-ul este registrul care indică întotdeauna spre baza stivei și prin care se va face delimitarea cadrelor de apel și accesarea elementelor corespunzătoare acestora.

In continuare este prezentat un exemplu pentru evaluarea expresiei

$$((x + y) * (x - y) * (x + z))/(y + z)$$

folosind stive. x, y, z sunt 3 numere de tip long declarate global.

```
 .data
x: .long 2
y: .long 1
z: .long 3
e: .space 4
.text
.global main
main:

movl y, %eax
addl z, %eax
pushl %eax
; %esp: (y + z)

movl x, %eax
addl z, %eax
pushl %eax
; %esp: (x + z), (y + z)

movl x, %eax
subl y, %eax
pushl %eax
; %esp: (x - y), (x + z), (y + z)

movl x, %eax
addl y, %eax
pushl %eax
; %esp: (x + y), (x - y), (x + z), (y + z)
```

```
popl %eax
popl %ebx
mull %ebx

pushl %eax
; %esp: (x + y) * (x - y), (x + z), (y + z)

popl %eax
popl %ebx
mull %ebx

pushl %eax
; %esp: (x + y) * (x - y) * (x + z), (y + z)

popl %eax
popl %ebx
divl %ebx

pushl %eax
; %esp: ((x + y) * (x - y) * (x + z)) / (y + z)

popl e

mov $1, %eax
xor %ebx, %ebx
int $0x80
```

Observatie: Atentie la comentariile din codul de mai sus! Ca si pana acum, in functie de ce environment utilizati, comentariile pot fi introduse ori cu `#`, ori cu `//`;

2 Proceduri in limbaj de asamblare

Daca o prelucrare trebuie facuta de mai multe ori in acelasi fel (eventual cu alte date), in loc sa rescriem grupul respectiv de instructiuni de mai multe ori in program putem sa-l incapsulam intr-o procedura si de fiecare data cand avem nevoie de el sa apelam procedura (eventual cu noile date transmise ca parametri).

In x86 procedurile nu au un mod specific de definire. Ele sunt simple blocuri de cod ape- late insa intr-o maniera specifica. Evident, ele trebuie declarate in zona **.text** si sunt identificate printr-un nume sub forma unei etichete.

2.1 Apelul unei proceduri

Mecanismul de control privind salturile intr-o procedura si revenirea in locul de unde a fost apelata este realizat prin doua instructiuni - **call** si **ret**.

```
call proc
```

unde **proc** este eticheta ce marcheaza inceputul procedurii. Prin aceasta instructiune se apeleaza procedura **proc** si se retine adresa de intoarcere (locul de unde procedura a fost apelata). De fapt, se produce o salvare pe stiva a valorii din %eip (instruction pointer) care retine fix adresa instructiunii urmatoare. In acelasi timp, %eip preia adresa primei instructiuni din procedura, realizandu-se astfel continuarea executiei din acel punct.

```
ret
```

Aceasta instructiune realizeaza un salt la adresa din varful stivei. Cum pe parcursul procedurii apelate toate elementele adaugate pe stiva sunt si eliminate, la final in varful stivei se va afla chiar adresa de retur retinuta prin instructiunea **call**.

O schita simplista este prezentata în continuare.

```
.data
.text

proc1:

    ; push-uri si pop-uri in numar egal

    ret ; stiva arata ca la intrare in procedura,
        ; in varf se afla deci adresa instructiunii mov 1, eax
        ; salt la adresa din varf

.global main
main:

    call proc1 ; este adaugata pe stiva adresa lui mov 1, eax
               ; salt la proc

    mov $1, %eax
    xor %ebx, %ebx
    int $0x80
```

2.2 Argumentele unei proceduri

In continuare, pentru a transmite argumentele unei proceduri, acestea vor fi depozitate tot pe stiva. Datorita conceptului pe care il implementeaza o stiva - LIFO (Last In, First Out) este necesar ca argumentele sa fie stocate pe stiva in ordine inversa pentru a putea fi accesate in ordine naturala. Pe cazul general, consideram urmatorul apel **proc(arg1, arg2, ..., argn)**. Schita pentru apel este urmatoarea.

```
pushl argn
pushl argn-1
...
pushl arg2
```

```
pushl arg1
call proc
popl %eax
popl %eax
...
popl %eax
; n pop-uri corespunzatoare celor n argumente
```

Observatie: Toate argumentele incarcate pe stiva, trebuie eliminate la iesirea din procedura.

Exemplu: Fie acum functia **add** avand declaratia **add(x,y,&s)** care realizeaza adunarea a doi intregi **x** si **y** si depoziteaza suma in **s**. Pana in acest punct, un program minimalist in care se cheama respectiva functie cu argumentele **2, 3**, respectiv adresa unei variabile din memorie, este prezentat mai jos.

```
.data
x: .long 2
y: .long 3
s: .space 4
.text

add:
    movl 4(%esp), %eax
    addl 8(%esp), %eax
    movl 12(%esp),%ebx
    movl %eax, 0(%ebx)

    ret

.global main
main:

    pushl $s
    pushl y
    pushl x
    call add
    popl %edx
    popl %edx
    popl %edx

    mov $1, %eax
    xor %ebx, %ebx
    int $0x80
```

Cerinta: Rulati cu debuggerul si observati evolutia stivei. Explicati indicii de accesare si modul prin care este transmis un parametru cu referinta.

Observatie: Indicii de accesare sunt crescatori, ordinea argumentelor fiind cea descrisa in enunt.

2.3 Crearea cadrului de apel

In exemplul anterior a fost folosit pentru accesare elementelor de pe stiva registrul %esp. Totusi aceasta modalitate de lucru poate deveni foarte rapid incomoda daca pe parcursul procedurii apelate este repetat un numar de instructiuni de **push** si **pop**. Dupa fiecare astfel de operatie ar trebui ca programatorul sa regandeasca asezarea de pe stiva si sa schimbe modul de accesare. Practic e posibil ca un acelasi element sa fie accesat o data ca **0(%esp)**, iar dupa adaugarea unei alte valori pe stiva sa fie accesat ca **4(%esp)**. Intrucat dorim sa mentinem constant modul in care ne raportam la valori de pe stiva, ne vom raporta la registrul **%ebp**, depre care stiam ca memoreaza baza stivei.

Anterior, a fost mentionat, de asemenea, ca pentru fiecare apel de functie se aloca o zona de memorie pe stiva care este golita la iesirea din functie. O astfel de zona se numeste **stack frame** si practic delimiteaza zona apartinand unei proceduri. Pentru procedura curenta, varful stack frame-ului va fi reprezentat de valoarea lui **%esp**. Am dori sa avem si o marcare a bazei acestei zone, iar numele ne trimite direct cu gandul la **%ebp**. Ar fi ideal ca la executia in procedura curenta, **%ebp** sa indice baza zonei proprii de memorii de pe stiva si nu baza intregii stive care nu ar fi de folos intrucat nu suntem interesati de toate **stack frame**-urile create anterior din diverse apeluri imbricate.

Pentru aceasta la fiecare creare de cadru de apel, registrul **%ebp** va fi modificat pentru a indica baza **stack frame**-ului curent. Totusi acesta nu este un pas suficient pentru ca la revenirea din procedura, ne-am dori ca **%ebp** sa indice catre baza **stack frame**-ului procedurii apelante. Atunci o solutie evidenta este sa memoram valoarea initiala la intrarea in procedura si sa o restauram la iesire pentru a se mentine in registrul **%ebp** valoarea corespunzatoare a cadrului in orice moment al executiei.

Astfel, un cadru corect de apel pentru schema din subsectiunea 2.1 este urmatorul:

```
.data
.text

proc1:
    pushl %ebp
    movl %esp, %ebp

    ; codul din corpul procedurii

    popl %ebp
    ret

.global main
main:

    call proc1


    mov $1, %eax
    xor %ebx, %ebx
    int $0x80
```

Asadar, vom rescrie exemplul din subsectiunea 2.2 raportat la noile cunostinte. Pentru aceasta sa vedem cum arata stiva.

Figura 2: Stiva program subsectiunea 2.2

In acest context, in cadrul functiei **add** putem adauga oricate elemente dorim in jos in stiva pentru ca **%ebp** va ramane fix, singurul care isi modifica valoarea fiind **%esp**. Astfel, programul va fi rescris ca in exemplul de mai jos.

Exemplu:

```
.data
x: .long 2
y: .long 3
s: .space 4
.text

add:
    pushl %ebp
    mov %esp, %ebp

    movl 8(%ebp), %eax
    addl 12(%ebp), %eax
    movl 16(%ebp),%ebx
    movl %eax, 0(%ebx)

    popl %ebp
    ret

.global main
main:

    pushl $s
    pushl y
```

```
pushl x

call add

popl %edx
popl %edx
popl %edx

mov $1, %eax
xor %ebx, %ebx
int $0x80
```

Cerinta: Rulati cu debuggerul si observati din nou evolutia stivei, precum si noii indici de accesare.

Observatie: Variabilele locale din proceduri se salveaza tot pe stiva in continuarea cadrului descris anterior.

2.4 Registrii callee si caller saved

Dupa cum bine stim, pe ahitectura x86 numarul de registri este limitat. Am considerat abuziv pana acum ca printr-o analogie cu C-ul, registrii pot fi vazuti ca niste variabile ale programului. In acest caz, urmand aceeasi analogie, vom considera cazul in care avem o variabila **x** declarata in **main** si o variabila cu acelasi numele intr-o procedura apelata. Este bine-cunoscut ca practic cele 2 variabile nu se influenteaza una pe cealalta. Modificarea variabilei **x** din procedura nu are niciun efect asupra celei din **main**. In acelasi mod, trebuie ca registrii modificati intr-o procedura sa nu produca efecte secundare asupra utilizarii lor din apelant. Dorim sa regasim dupa intoarcerea din procedura, in toti registri folositi valorile depozitate inainte de apel.

Pentru aceasta avem din nou ca instrument principal stiva. Astfel in functie de locul in care sunt salvate valorile registrilor, exista 2 tipuri de registri:

- **callee-saved : %ebx, %esp, %ebp, %esi, %edi**. Valorile acestor registri se garanteaza a fi restaurate de catre procedura, adica acestea trebuie salvate in zona de variabile locale de catre procedura.
- **caller-saved : %eax, %ecx, %edx**. Nu este garantata restaurarea lor. Apelantul trebuie sa salveze aceste valori inainte de incarcarea argumentelor functiei daca doreste sa regaseasca valorile initiale la iesirea din functie.

In cele din urma, asadar, un cadru de apel va arata ca in figura 3

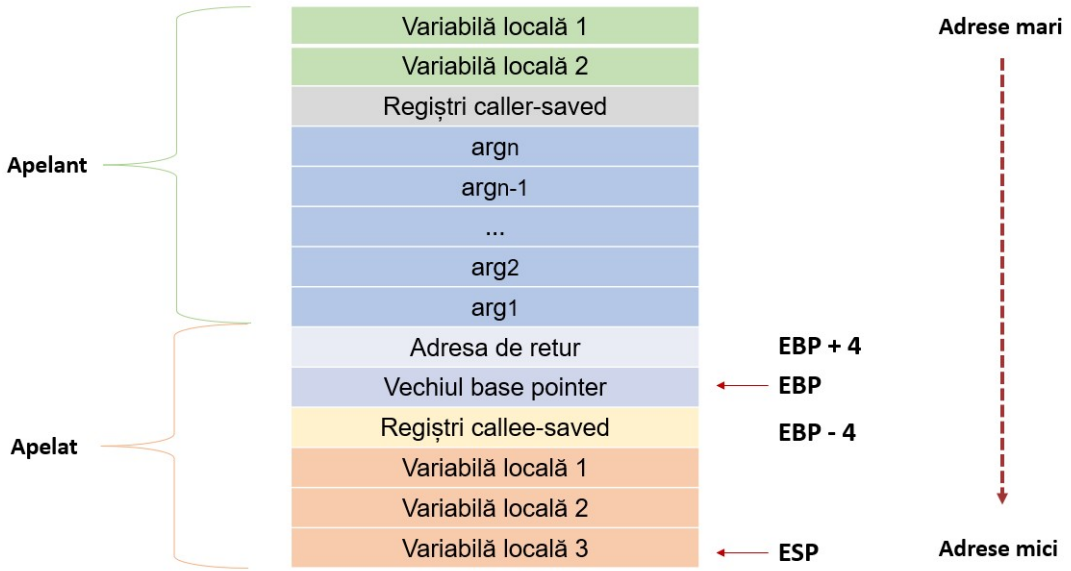

Figura 3: Cadru de apel

Exemplu: Sa luam acum acelasi cod pentru apelul functiei **add** si sa presupunem ca dupa apel exista o bucata de cod suplimentara care se asteapta sa reagaseasca in registrii **%eax** si **%ebx** valorile de dinainte de apel. O rescriere este prezentata in continuare.

```
.data
x: .long 2
y: .long 3
s: .space 4
.text

add:
    pushl %ebp
    mov %esp, %ebp

    pushl %ebx

    movl 8(%ebp), %eax
    addl 12(%ebp), %eax
    movl 16(%ebp),%ebx
    movl %eax, 0(%ebx)
```

```
    popl %ebx

    popl %ebp
    ret

.global main
main:

    movl $5, %eax
    movl $6, %ebx

    pushl %eax
    pushl $s
    pushl y
    pushl x
    call add

    popl %edx
    popl %edx
    popl %edx
    popl %eax

    eticheta:

    ; cod suplimentar

    mov $1, %eax
    xor %ebx, %ebx
    int $0x80
```

Cerinta: Rulati cu debuggerul si observati din nou evolutia stivei, precum si valorile registrilor %eax, %ebx inainte de inceperea bucatii de cod suplimentare.

2.5 Returnarea valorilor

Pana acum, am considerat numai proceduri care nu returnau nicio valoare. In cazul in care se doreste returnare, valorile vor fi depozitate in aceasta ordine in functie de numarul lor in **%eax, %ecx, %edx** si ulterior prin varful stivei.

Observatie: Registrii prin care se face returul sunt exact registrii care nu sunt restaurati de apelat. Totusi, si ei pot fi restaurati la un anumit punct in program daca valorile initiale erau necesare apelantului. Asadar, valorile returnate trebuie utilizate (sau memorate) inaintea unei astfel de instructiuni de restaurare a vechii valori.

Cerinta: Modificati ultimul program astfel incat procedura **add** sa aiba doar doi parametri si sa returneze suma in programul principal. Suma va fi ulterior incarcata in memorie la adresa lui **s**.

3 Apelul functiilor din C

Pentru a putea interactiona intr-un program dezvoltat in limbaj de asamblare cu functii din limbajul `C`, putem utiliza, in loc de *link*-area prezentata in primul suport de laborator, compilatorul `gcc`. In aceasta maniera, putem folosi functiile predefinite in libraria standard, printre care si cele mentionate anterior, si anume `printf`, `scanf` si `fflush`.

3.1 Apelul functiei `printf`

In limbajul `C`, functia `printf` este definita ca avand un numar variabil de argumente, primul fiind formatul `string`-ului de afisat. De exemplu, apeluri valide sunt urmatoarele:

```
printf("Numarul de afisat este: %ld\n", x);
printf("Numarul de afisat este: %d\n", x);
printf("Suma numerelor %hu si %hu este %hu\n", x, y, sum);
printf("Numele studentului cu cea mai mare medie este %s\n", nume);
```

Doua dintre formatele utile pe care le vom utiliza sunt

- `%d` - pentru afisarea int-urilor;
- `%ld` - pentru afisarea long-urilor;
- `%hu` - pentru afisarea short int-urilor (word-urilor);
- `%s` - pentru afisarea sirurilor de caractere.

Observatie! Long-urile și int-urile din C pe 32 de biți sunt echivalentul unui long in asamblare (ocupă 4 octeți)

Numarul argumentelor pe care le ia `printf` este dat de cate formate speciale utilizam in cadrul primului argument - de exemplu, pentru ca in sirul pentru afisarea sumei de la exemplul anterior avem trei aparitii ale lui `%ld`, inseamna ca numarul de argumente va fi egal cu 1 (sirul de format) + cate un argument pentru fiecare valoare care trebuie completata (3) = 4.

Pentru a apela functia `printf` intr-un program dezvoltat in limbaj de asamblare, vom proceda astfel: (presupunem ca urmatorul fisier este denumit `call_printf.s`):

```
.data
    x: .long 23
    formatString: .asciz "Numarul de afisat este: %ld"
.text

.global main

main:
    ; incarcam argumentele in ordinea inversa limbajului C
    ; pentru printf("Numarul de afisat este: %ld", x);
    ; incarcam x, apoi sirul de format
    pushl x
    pushl $formatString
```

```
    ; apelam functia printf
    call printf

    ; pentru fiecare push facut, facem un pop
    ; alegand un registru pe care nu-l utilizam
    popl %ebx
    popl %ebx

    movl $1, %eax
    xorl %ebx, %ebx
    int $0x80
```

Observatii importante:

- pentru ca `printf` acceseaza argumentele in ordinea naturala - mai intai formatul sirului de afisat, apoi argumentele, in ordinea in care apar si in format, inseamna ca le vom incarca in **ordine inversa** pe stiva;
- formatul sirului **nu** se da ca valoare, ci **ca adresa in memorie**, de aceea trebuie ca numele lui sa fie prefixat de simbolul `$`;
- pentru a face `pop`, putem utiliza orice registru care nu contine, momentan, o valoare pe care vrem sa o pastram - `pop` modifica accesul pe care il are registrul `%esp` (*stack pointer*-ul) asupra stivei, si descarca valoarea actuala din *top* in registrul pe care noi il precizam. In acest caz, am ales sa facem descarcarea in `%ebx`, fiind echivalent orice alt registru;
- instructiunea `xorl %ebx, %ebx` este echivalenta cu instructiunea `movl $0, %ebx`.

Pentru a compila programul, vom utiliza compilatorul `gcc`:

```
gcc -m32 call_printf.s -o call_printf
./call_printf
```

Cel mai probabil, in acest moment nu vom avea o valoare efectiva afisata la **standard output**, din cauza modului in care sunt implementate sistemele de operare. Pentru a obliga afisarea *buffer*-ului curent de memorie, vom utiliza functia `fflush` cu argumentul `NULL`. Apelul este foarte simplu, trebuie sa incarcam pe stiva `NULL`, adica valoarea `0`, si sa o descarcam, din nou, intr-un registru pe care nu-l avem momentan completat cu o valoare utila - sa presupunem ca tot `%ebx`. Apelul va fi:

```
pushl $0
call fflush
popl %ebx
```

In acest moment, exemplul complet va avea urmatoarea forma:

```
.data
    x: .long 23
    formatString: .asciz "Numarul de afisat este: %ld"
.text
```

```
.global main

main:
    pushl x
    pushl $formatString
    call printf
    popl %ebx
    popl %ebx

    pushl $0
    call fflush
    popl %ebx

    movl $1, %eax
    xorl %ebx, %ebx
    int $0x80
```

Vom compila iar:

```
gcc -m32 call_printf.s -o call_printf
./call_printf
```

Care, de aceasta data, va afisa corect pe ecran mesajul

```
Numarul de afisat este: 23
```

Observatie. Pentru a standardiza modul in care vom face *call*-urile catre `printf`, vom avea mereu un `printf` urmat de un `fflush`.

3.2 Apelul functiei `scanf`

Vom pleca, din nou, de la modul in care aceasta functie este apelata in limbajul `C`, si anume:

```
scanf("%ld", &x);
scanf("%ld %ld", &x, &y);
```

Desi apelul seamana foarte mult cu cel pe care il utilizam pentru `printf`, observam ca, de aceasta data, argumentele nu mai sunt date prin valoarea propriu-zisa - nu apelam `x`, ci `&x`. Acest lucru trebuie sa se reflecte si in limbajul de asamblare. Pentru aceasta, avem simbolul `$`, care ne da acces la locatia de memorie a unei variabile declarate in sectiunea `.data`.

Vom demonstra aceste lucruri prin intermediul codului de mai jos, presupus salvat intr-un fisier de forma `call_scanf.s`, pentru citirea unui intreg de la tastatura.

```
.data
    x: .space 4 ; nu stim momentan valoarea citita, dar stim cat ocupa
    formatString: .asciz "%ld"
.text

.global main
```

```
main:
    ; incarcam in stiva NU x, ci ADRESA lui x
    ; adica nu x, ci &x
    pushl $x
    ; incarcam adresa formatului
    pushl $formatString
    call scanf
    ; descarcam ce am incarcat in stiva
    popl %ebx
    popl %ebx

    movl $1, %eax
    xorl %ebx, %ebx
    int $0x80
```

3.3 Un exemplu de citire si de afisare a unui graf neorientat

Pentru a demonstra utilizarea pentru `printf`, `scanf` si `fflush`, vom prezenta citirea unui graf neorientat, dat prin numarul de varfuri, numarul de muchii si muchiile acestuia.

```
.data
        matrix: .space 1600
        columnIndex: .space 4
        lineIndex: .space 4
        n: .space 4
        nrMuchii: .space 4
        index: .space 4
        left: .space 4
        right: .space 4
        formatScanf: .asciz "%ld"
        formatPrintf: .asciz "%ld "
        newLine: .asciz "\n"
.text

.global main

main:
        pushl $n
        pushl $formatScanf
        call scanf
        popl %ebx
        popl %ebx

        pushl $nrMuchii
        pushl $formatScanf
        call scanf
        popl %ebx
        popl %ebx
```

```
    ; simulam, de fapt
    ; for (long index = 0; index < nrMuchii; index++)
    ; {
    ;       scanf("%ld", &left);
    ;       scanf("%ld", &right);
    ;       matrix[left][right] = 1;
    ;       matrix[right][left] = 1;
    ; }
        movl $0, index
et_for:
        movl index, %ecx
        cmp %ecx, nrMuchii
        je et_afis_matr

        pushl $left
        pushl $formatScanf
        call scanf
        popl %ebx
        popl %ebx

        pushl $right
        pushl $formatScanf
        call scanf
        popl %ebx
        popl %ebx

        ; trebuie sa completez
        ; matrix[left][right] = 1
        ; indexul este left * n + right

        movl left, %eax
        movl $0, %edx
        mull n
        ; %eax := left * n
        addl right, %eax
        ; %eax := left * n + right

        lea matrix, %edi
        movl $1, (%edi, %eax, 4)

        ; trebuie sa completez matrix[right][left] = 1
        ; indexul este right * n + left

        movl right, %eax
        movl $0, %edx
        mull n
        ; %eax := right * n
```

```
        addl left, %eax
        ; %eax := right * n + left

        lea matrix, %edi
        movl $1, (%edi, %eax, 4)

        incl index
        jmp et_for

et_afis_matr:
        movl $0, lineIndex
        for_lines:
                movl lineIndex, %ecx
                cmp %ecx, n
                je et_exit

                movl $0, columnIndex
                for_columns:
                        movl columnIndex, %ecx
                        cmp %ecx, n
                        je cont

                        ; afisez matrix[lineIndex][columnIndex]
                        ; indexul este lineIndex * n + columnIndex
                        movl lineIndex, %eax
                        movl $0, %edx
                        mull n
                        addl columnIndex, %eax
                        ; %eax = lineIndex * n + columnIndex

                        lea matrix, %edi
                        movl (%edi, %eax, 4), %ebx

                        pushl %ebx
                        pushl $formatPrintf
                        call printf
                        popl %ebx
                        popl %ebx

                        pushl $0
                        call fflush
                        popl %ebx

                        incl columnIndex
                        jmp for_columns

        cont:
                movl $4, %eax
```

```
                movl $1, %ebx
                movl $newLine, %ecx
                movl $2, %edx
                int $0x80

                incl lineIndex
                jmp for_lines


et_exit:
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

Nota: Testati-va cunostintele acumulate pana acum facand testele de laborator 6.1, 6.2 si 6.3 pe care le gasiti online la sectiunea laboratoare.

4 Manipularea numerelor în virgulă mobilă în x86 AT&T

Arhitectura x86 suportă operații pe numere în virgulă mobilă, utilizând două abordari, **FPU** (*Floating Point Unit*) si **SSE** (*Streaming SIMD Extensions*). Dintr-o perspectivă istorică, FPU a fost prima introdusă, iar ulterior a existat o extindere prin introducerea convenției SSE.

Numerele în virgulă mobilă sunt reprezentate conform standardului *IEEE 754*, iar in cadrul laboratorului vom lucra doar cu formatul `float`, pe 32 biți (arhitectura suportă si un format `double`, pentru mai multă precizie, reprezentat pe 64 biți). În formatul `float`, avem 1 bit pentru semn, 8 biți pentru exponent, respectiv 23 biți pentru mantisă, iar codificarea unei valori este dată de formula $\text{Val} := (-1)^S \cdot 2^{\text{E}-\text{Bias}} \cdot (1.\text{M})$.

4.1 FPU

FPU a fost inițial implementată ca un coprocesor hardware (de exemplu, coprocesorul 8087 pentru procesorul 8086) și mai târziu integrată direct în procesoare. Aceasta utilizează o stivă internă (stack-based) pentru operații. Putem considera aceste locații ca niște registrii de la `st(0)` la `st(7)`, care sunt organizați într-o stivă, iar instrucțiunile **FPU** operează implicit pe aceasta stivă.

In ceea ce priveste abordarile, respectiv registrii utilizati, pentru **FPU** avem registrii de la `st(0)` la `st(7)`, care sunt organizati intr-o stiva, iar instructiunile **FPU** opereaza implicit pe aceasta stiva, iar pentru **SSE** avem registrii de la `xmm0` la `xmm7`.

In cele ce urmeaza, prezentam câteva instructiuni pe care le putem utiliza pentru a putea manipula numerele în virgulă mobilă.

Instructiune	Operanzi	Scurta explicatie
`flds`	`mem`	Incarca un `float` in stiva FPU, in `st(0)`
`fldl`	`mem`	Incarca un `double` in stiva FPU, in `st(0)`
`fstps`	`mem`	Stocheaza un `float` din `st(0)` in memorie
`fstpl`	`mem`	Stocheaza un `double` din `st(0)` in memorie

De asemenea, avem la dispozitie functii matematice in `libm`, biblioteca pe care o putem specifica in comanda de compilare, prin flag-ul `-lm`. (de exemplu `gcc -m32 float.s -o float -no-pie -lm`) Exemplu de funcții pe care le putem utiliza: `logf`, `sqrtf`, `expf`. Convenția de apel se respectă, argumentele se incarcă pe stivă în ordine inversă, iar rezultatul funcției este preluat din `st(0)`. Aceste funcții sunt exact motivul pentru care FPU este încă folosit întrucât ele sunt implementate conform convenției acesteia mai vechi. FPU este păstrat deci pentru compatibilitatea cu bibliotecile implementate pe acest standard.

Exemplu pentru calculul logaritmului natural:

```
.data
    number: .float 2.718281828
    logResult: .space 4
.text

.global main

main:
```

```
    flds number     // incarcam valoarea pe stiva FPU
    subl $4, %esp
    fstps 0(%esp)   // efectuam un push pentru a respecta conventia de apel

    call logf

    fstps logResult // am primit rezultatul in st(0), il descarcam si il salvam in logResult
    addl $4, %esp   // pop pe stiva

et_exit:
    movl $1, %eax
    xorl %ebx, %ebx
    int $0x80
```

Rulam acest program (din fisierul numit, de exemplu, `float1.s`) cu `gcc -m32 float1.s -o float1 -no-pie -lm`. Utilizam `gdb` pentru a verifica daca s-a efectuat calculul corect: `b et_exit`, `run`, `print (float) logResult`. Observam ca obtinem valoarea `0.99999994`.

4.2 SSE

SSE a fost introdus ca o extindere modernă pentru a rezolva limitările FPU și pentru a oferi performanță sporită în anumite scenarii. Este destinat operațiilor pe date vectoriale (ex. procesare grafică, audio, video, fizică pentru jocuri), permițând procesorului să execute aceleași operații pe mai multe date simultan.

Sunt folosiți regiștri vectoriali (de exemplu, regiștrii xmm de 128 biți) care pot opera pe mai multe valori simultan (cum ar fi 4 valori float de 32 de biți în același timp). Vom omite aceste extinderi ale lor în cadrul laboratorului folosind regiștrii de la `xmm0` la `xmm7` doar pentru a reține o singură valoare scalară.

Din nou, convenția de apel se respectă, argumentele se încarcă pe stivă în ordine inversă, iar de data aceasta rezultatul funcției este preluat din `xmm0`. Procedurile nou adăugate (chiar și de voi) se pot mula pe aceasta convenție mai nouă.

Pentru **SSE** vom avea în vedere următoarele câteva instrucțiuni:

Instructiune	Operanzi	Scurta explicatie
`movss`	`src, dest`	Muta un scalar **float** de la **src** la **dest**
`movsd`	`src, dest`	Muta un scalar **double** de la **src** la **dest**
`addss`	`src, dest`	Aduna `float` din `src` cu `dest` si stocheaza in `dest`
`subss`	`src, dest`	Scade `src` din `dest` si stocheaza in `dest`
`mulss`	`src, dest`	Inmulteste `src` cu `dest` si stocheaza in `dest`
`divss`	`src, dest`	Imparte `dest` la `src` si stocheaza in `dest`
`sqrtss`	`src, dest`	Calculeaza `dest := sqrt(src)`
`cvtsi2ss`	`src, dest`	Converteste un `long` in `float` si pune rezultatul in `dest`
`cvtss2sd`	`src, dest`	Converteste un `float` in `double` si pune rezultatul in `dest`
`maxss`	`src, dest`	Calculeaza maximul in `dest`
`minss`	`src, dest`	Calculeaza minimul in `dest`

Deși în cadrul laboratorului vom folosi în principal numere `float`, în tabelul anterior au fost prezentate și câteva instrucțiuni pentru `double`. Acestea vor fi utile în contextul realizării unei afișări, întrucât deși conform convențiilor format string-ul `"%f"` ar trebui să aștepte ca argument un `float`, implementarea funcției `printf` cere întotdeauna ca argument un `double`.

Vom prezenta în continuare o secvență de cod pentru realizarea afișării rezultatului `logResult` calculat în exemplul anterior:

```
.data
    number: .float 2.718281828
    logResult: .space 4
    fs: .asciz "%f\n"
.text
    ...
et_printf:
    cvtss2sd logResult, %xmm0   // conversia la double
    sub $12, %esp               // 8 bytes pentru double + 4 pentru format string
    movsd %xmm0, 4(%esp)        // punem pe stiva double-ul
    movl $fs, 0(%esp)           // punem pe stiva adresa format string-ului
    call printf                 // apel la printf
    add $12, %esp               // scoatem de pe stiva
```

Oferim și un exemplu pentru determinarea mediei aritmetice a elementelor dintr-un array de `float`-uri:

```
.data
    v: .float 1.0, 2.0, 3.0, 4.0
    n: .long 4
    result: .space 4
    mem: .space 4
.text

.global main

main:
    lea v, %edi                 // adresa array-ului
    xorl %ecx, %ecx             // pornim de la indexul 0
    movl $0, mem                // utilizam memorie intermediara
    movss mem, %xmm0            // initializam suma cu 0

et_loop:
    cmpl n, %ecx
    je avg

    movss (%edi,%ecx,4), %xmm1  // incarcam elementul din array
    addss %xmm1, %xmm0          // il adaugam la suma

    incl %ecx
```

```
    jmp et_loop

avg:
    cvtsi2ss %ecx, %xmm1        // convertim numarul de elemente la float
    divss %xmm1, %xmm0          // impartim si salvam in xmm0 rezultatul

    movss %xmm0, result

et_exit:
    movl $1, %eax
    xorl %ebx, %ebx
    int $0x80
```

Pentru a inspecta valoarea, consideram ca fisierul se numeste `float2.s`, si rulam `gcc -m32 float2.s -o float2 -no-pie -lm`, iar cu `gdb`: `b et_exit`, `run`, `print (float) result` si observam ca obtinem valoarea `2.5`.

Exercitiu. Considerati ca aveti un *array* de `float` care stocheaza probabilitatile $\{p_i\}_{i\in\{1,\dots,n\}}$ a n evenimente. Determinati **entropia** acestei multimi. Implementati aceasta problema utilizand o procedura cu signatura `float entropy(float *probabilities, float n)`.

5 Exerciții

1. (a) Sa se defineasca procedura perfect(x), cu x numar natural. Un numar este perfect daca este egal cu suma divizorilor sai pana la jumatate. Exemplu: $6 = 1 + 2 + 3$; $28 = 1 + 2 + 4 + 7 + 14$;

   (b) Se dau de la tastatura un intreg **n** si un vector cu **n** elemente. Sa se afiseze pe ecran numarul de elemente perfecte.

2. Sa sa implementeze procedura x86 cu acelasi efect ca procedura C **memcpy**:

   ```
   void *memcpy(void *dest, const void *src, size_t n);
   ```

   care copiaza **n** octeti de la adresa **src** la adresa **dest** si returneaza **dest**.

3. Scrieti o procedura ce implementeaza functia C **atoi**:

   ```
   int atoi(const char *s);
   ```

   care returneaza intregul a carui reprezentare externa zecimala este continuta in stringul **s** sau 0 daca stringul nu poate fi convertit.

4. Translatati in limbaj de asamblare x86 urmatorul program C:

   ```
   #include<stdarg.h>

   void aduna(long *a, long n, ...){
       register long i;
       va_list l;
       va_start(l,n);
       *a=0;
       for(i=0;i<n;++i) *a+=va_arg(l, long);
       va_end(l);
   }

   long s, s1;

   void main(){
       aduna(&s,3,1,2,3);     /* obtinem s=6 */
       aduna(&s1,2,10,20);    /* obtinem s1=30 */
   }
   ```

5. Sa se implementeze un program care sa calculeze functia $f(x) = 2g(x)$, unde $g(x) = x + 1$.

6. Sa se implementeze procedura **proc(x)**, $x > 1$, cu definitia:

   ```
   void proc(long x)  {
       printf("\%ld ",x);
       if (x != 0)
           proc(x-1);
   }
   ```

7. Sa se transcrie in limbaj de asamblare x86:

```
long aduna(long a, long b) {
    return a+b;
}

void iteratie(long *a, long *b) {
    long c;
    c=aduna(*a, *b);
    (*a)=(*b); (*b)=c;
}

void main() {
    long n=5,x=1,y=1,z;
    register long i;
    for(i=2;i<n;++i) iteratie(&x,&y);
    z=y;
}
```

8. Sa se implementeze o procedura care calculeaza recursiv factorialul unui numar.

9. Scrieti un program care sa calculeze al n-lea termen (t_n) din sirul lui Fibonacci (folosind recursivitate). Afisati la final un text de forma *Al n-lea element din sirul lui Fibonacci este x.* Vom considera primele 2 elemente $t_0 = 0$ si $t_1 = 1$. Exemplu: $n = 5$ => Al 5-lea termen din sirul lui Fibonacci este 5.

10. Se introduc de la tastatura n si k ($k \leq n$). Sa se scrie o functie $C(n, k)$ recursiva ce calculeaza combinari de n luate cate k dupa formula:

$$C(n,k) = \begin{cases} 1 & \text{daca } k = 0 \\ 1 & \text{daca } k = n \\ C(n,k) = C(n-1,k) + C(n-1,k-1) & \text{altfel.} \end{cases}$$

6 Resurse suplimentare

Pentru mai multe detalii despre GNU Assembly va recomandam cartea *Professional Assembly Language* de Richard Blum si in special urmatoarele capitole:

- Chapter 7: Using numbers subcapitolele SIMD Integers, Floating-Point Numbers;
- Chapter 9: Advanced Math Functions;
- Chapter 11: Using Functions;
- Chapter 12: Using Linux System Calls;
- Chapter 13: Using Inline Assembly;
- Chapter 15: Optimizing Routines;
- Chapter 17: Using Advanced IA-32 Features.